Práctica 1: Búsquedas con trayectorias simples
Selección de características
Primero el mejor, enfriamiento simulado y búsqueda tabú básica

Alejandro García Montoro
76628233F, agarciamontoro@correo.ugr.es

Grupo de los viernes a las 17.30

Curso 2015 - 2016

Índice

1. Descripción del problema

La selección de características es una técnica muy usada en problemas de aprendizaje automático.

El aprendizaje automático, visto de una forma muy general, tiene como objetivo clasificar un conjunto de objetos —modelador por una serie de atributos— en clases.

Esta clasificación se aprende desde los datos, pero la selección de los atributos que definen la modelización del objeto puede no ser la más apropiada: en ocasiones hay atributos superfluos o demasiado ruidosos que sería conveniente eliminar. Además, cuantos menos atributos definan un objeto, más rápido y preciso será el aprendizaje. Es aquí entonces donde aparece la pregunta que guia todo este trabajo: ¿cómo identificar los atributos que mejor aprendizaje promueven?

La respuesta a esta pregunta pasa por la selección de características, cuyo objetivo es reducir la definición de un objeto a una serie de características que faciliten el aprendizaje.

La idea es entonces la siguiente: dado un conjunto de m objetos definidos por un conjunto C de n características y considerada un modelo de aprendizaje f que intenta aprender la clasificación de estos objetos encontrar el subconjunto $C' \subset C$ que maximiza del modelo f.

Así, vemos claramente que el tamaño de caso de nuestro problema es n, el número de características, y que el objetivo está bien definido: eliminar aquellas características que o bien empeoren la bondad de f o bien sean innecesarias.

Con todos estos elementos definidos, podemos pasar a analizar las metaheurísticas consideradas.

2. Metaheurísticas

2.1. Introducción

Los algoritmos considerados para resolver el problema son los siguientes:

- *Best first local search*
- *Simulated annealing*
- *Short-term memory tabu search*

Además, compararemos estas metaheurísticas con el algoritmo voraz *Sequential forward selection*.

Estas tres metaheurísticas reúnen las condiciones necesarias para resolver el problema: el espacio de soluciones de nuestro problema puede ser analizado mediante las estructuras de generación de vecinos y los criterios de aceptación que utilizan estos algoritmos. Veamos con un poco más de detalle los aspectos comunes a las metaheurísticas implementadas:

Datos de entrada

Todos los algoritmos considerados reciben un conjunto de entrenamiento cuyos objetos tienen la siguiente estructura:

$$(s_1, s_2, \ldots, s_n, c)$$

donde $(s_1, s_2, \ldots, s_n)$ es el conjunto de valores de los atributos que definen el objeto y c la clase a la que pertenece.

Esquema de representación

El espacio de soluciones S de nuestro problema es el conjunto de todos los vectores s de longitud n —el número de características— binarios; es decir:

$$S = \{s = (s_1, s_2, \ldots, s_n) / s_i \in \{0, 1\}\ \forall i = 1, 2, \ldots, n\}$$

La posición i-ésima de un vector $s \in S$ indicará la inclusión o no de la característica i-ésima en el conjunto final C'.

Función objetivo

La finalidad de las metaheurísticas será maximizar la función objetivo siguiente:

$$\begin{aligned} f &: S \to [0, 100] \\ s &\mapsto f(s) = \text{Acierto del 3-NN sobre s} \end{aligned}$$

$f(s)$ es, por tanto, la tasa de acierto del clasificador 3-NN producido a partir de la solución s.

El clasificador 3-NN es una particularización del clasificador k-NN, que mide la distancia de la instancia considerada a todos los demás objetos en el conjunto de datos de entrenamiento y le asigna la clasificación mayoritaria de entre los k vecinos más cercanos; esto es:

Pseudocódigo 1 Clasificador k-NN

```
1: function k-NN(instance, trainingData)
2:     distances ← euclideanDistance(instance, trainingData)
3:     neighbours ← getClosestNeighbours(distances)
4:     classification ← mostVotedClassification(neighbours)
5:     return classification
```

Entorno de soluciones

Dada una solución $s \in S$, el entorno de soluciones vecinas a s es el conjunto

$$E(s) = \{s' \in S / s' - s = (0, \ldots, 0, \underbrace{1}_{i}, 0, \ldots, 0), i \in \{1, 2, \ldots, n\}\}$$

es decir, $E(s)$ son las soluciones que difieren de s en una única posición. Es evidente entonces que el conjunto $E(S)$ tiene siempre exactamente cardinal igual a n.

El operador de generación de vecino de la solución s es entonces como sigue:

Pseudocódigo 2 Operador de generación de vecino

```
1: function FLIP(solution, feature)
2:     s' ← solution
3:     s'[feature] ← (s'[feature] + 1) mod 2
4:     return s'
```

Criterios de parada

Aunque los criterios de parada dependerán de la metaheurística considerada —en general se parará cuando no se encuentre mejora en el entorno—, en todos los algoritmos pararemos necesariamente tras llegar a las 15000 evaluaciones con el clasificador 3-NN sobre las soluciones generadas.

2.2. Búsqueda local primero el mejor

El primer algoritmo considerado es una búsqueda local de primero el mejor muy sencilla. El pseudocódigo de todo el procedimiento es el siguiente:

Pseudocódigo 3 Búsqueda local primero el mejor

```
1: function BESTFIRST(train, target)
2:     s ← genInitSolution()
3:     besScore ← score(s, train, target)
4:     improvementFound ← True
5:     while improvementFound do
6:         improvementFound ← False
7:         for f ← genRandomFeature(s) do          ▷ Without replacement
8:             s'← genNeighbour(s,f)
9:             score ← score(s', train, target)
10:            if score > bestScore then
11:                bestScore ← score
12:                s ← s'
13:                improvementFound ← True
14:                break
15:    return s, bestScore
```

El método de exploración del entorno es el siguiente: dada una solución s, escogemos una característica al azar, aplicamos el operador $flip$ para obtener una solución vecina y medimos su bondad; si es mejor que s, nos quedamos con ella como mejor solución y volvemos a empezar; si no, tomamos otra característica al azar —sin repetir— y seguimos el proceso.

Pararemos el algoritmo si y sólo si, al haber explorado el entorno completo de la solución actual, ninguna de las soluciones vecinas es mejor. Estaremos entonces ante un máximo —probablemente local— y el algoritmo no puede mejorar la solución.

2.3. Enfriamiento simulado

La metaheurística de enfriamiento simulado es un ejemplo de estrategia de búsqueda por trayectorias simples.

La idea de este algoritmo es mantener una variable de temperatura, de manera que cuando esta sea alta la diversificación en el entorno de búsqueda será muy amplia —podremos pasar a zonas peores, explorando así muchas zonas diferentes del espacio de búsqueda y evitando máximos locales— y conforme tiene a la temperatura final, se procede a una fase de intensificación sobre una parte del espacio.

En este caso, además, debemos almacenar siempre la mejor solución, de manera que aunque al final intensifiquemos sobre una zona pobre, si al principio la diversificación fue exitosa, tengamos más posibilidades de obtener una solución buena.

Antes de entrar en los detalles, veamos primero el pseudocódigo del procedimiento en general:

Pseudocódigo 4 Enfriamiento simulado

```
1: function SIMULATEDANNEALING(train, target)
2:     s ← genInitSolution()
3:     bestSolution ← s
4:     currentScore, bestScore ← score(s, train, target)
5:     t ← t_0
6:     while t > t_f and neighboursAccepted > 0 and eval < 15000 do
7:         neighboursAccepted ← 0
8:         while not cooling needed do
9:             f ← genRandomFeature(s)                ▷ With replacement
10:            s'← genNeighbour(s,f)
11:            newScore ← score(s', train, target)
12:            Δ = currentScore - newScore
13:            if Δ < 0 or acceptWorseSolution = True then
14:                currentScore ← newScore
15:                acceptedNeighbourgs++
16:                if currentScore > bestScore then
17:                    bestScore, bestSolution ← currentScore, s
18:        t ← coolingScheme(t)
19:    return s, bestScore
```

En este algoritmo hay tres cuestiones que debemos detallar: la generación de la temperatura inicial, la condición que se debe de cumplir para proceder al enfriamiento y la determinación de la aceptación de una solución peor que la actual.

Temperaturas inicial y final

El esquema para la generación de la temperatura inicial es el siguiente:

$$T_0 = \frac{\mu f(s_0)}{-\log(\phi)}$$

donde $f(s_0)$ es la tasa de clasificación de la función objetivo con la solución inicial y donde se ha tomado $\mu = \phi = 0{,}3$

Además, se ha tomado el siguiente valor para la temperatura final, que controla el fin del algoritmo:

$$T_f = 0{,}001$$

Condición para el enfriamiento

En el bucle interno del algoritmo se generan soluciones vecinas de la actual, aceptándolas o no dependiendo de su bondad y de una función probabilística que ahora describiremos y que depende de la temperatura.

Por tanto, en este bucle hay que controlar la condición que determinará cuándo se sale de él y se procede al enfriamiento, pasando así a una nueva fase con una función probabilística distinta. Esta condición es la siguiente: que el número de vecinos generados y de vecinos aceptados sean menores que unos máximos predeterminados —dependientes del tamaño del problema—. Estos máximos se calculan de la siguiente manera:

$$\text{Máximo de vecinos generados} = 10n$$

$$\text{Máximo de vecinos aceptados} = \frac{1}{10}\text{Máximo de vecinos generados}$$

Aceptación de soluciones peores que la actual

La potencia del enfriamiento simulado se encuentra en poder aceptar soluciones peores que la actual, de manera que se explore de una forma más amplia el espacio de búsqueda y se reduzca la posibilidad de quedar atrapado en un máximo local.

En este algoritmo hemos considerado el esquema de Cauchy modificado, donde la temperatura en la iteración $k+1$, dependiente de la iteración k y de una constante β, es la siguiente:

$$T_{k+1} = \frac{T_k}{1 + \beta T_k}$$

donde la constante β está definida como sigue:

$$\beta = \frac{T_0 - T_f}{M T_0 T_f}$$

con el siguiente valor de M:

$$M = \frac{\text{máximo de iteraciones permitidas}}{\text{máximo de vecinos generados}} = \frac{15000}{10n} = \frac{1500}{n}$$

2.4. Búsqueda tabú básica

La búsqueda tabú es una herramienta muy potente para muchos problemas, incluido el que estamos considerando. La idea es hacer una búsqueda local manteniendo una serie de movimientos prohibidos, aceptando siempre —incluso aunque sea peor— la mejor solución vecina. Las soluciones mejores que la mejor solución encontrada están eximidas de la prohibición determinada por la lista tabú; a la condición que determina las soluciones eximidas la llamaremos criterio de aspiración, y en este caso es la ya descrita: que la solución considerada sea mejor que la mejor solución encontrada hasta ahora en todo el algoritmo. El pseudocódigo del procedimiento implementado es el siguiente:

En este caso, el entorno de la solución está restringido a 30 elementos, y se genera de forma aleatoria tomando 30 características diferentes con las que poder aplicar el operador de generación de vecino.

Pseudocódigo 5 Búsqueda tabú

```
1: function TABUSEARCH(train, target)
2:     s ← genInitSolution()
3:     bestSolution ← s
4:     currentScore, bestScore ← score(s, train, target)
5:     t ← t_0
6:     while there was some change and evaluations < 15000 do
7:         for f ← sampleFeature(s) do        ▷ Sample 30 different features
8:             s'← genNeighbour(s,f)
9:             currentScore ← score(s', train, target)
10:            if f is in tabu list then
11:                if currentScore > bestScore then   ▷ Aspiration criterion
12:                    bestScore, bestSolution ← currentScore, s
13:            else if currentScore > bestLocalScore then        ▷ Best local
14:        if there was some changed feature then
15:            Pop last feature from tabu list and push changed feature
16:            s ← s'
17:    return s, bestScore
```

Lista tabú

El manejo de la lista tabú podemos especificarlo con más concreción como sigue:

- Inicialización: la lista tabú inicial será una lista vacía de tamaño $\frac{n}{3}$.
- Añadir elementos: cada vez que añadamos una característica prohibida a la lista tabú, debemos eliminar la última —aquella que lleva ya $\frac{n}{3}$ iteraciones en la lista—, con una estrategia FIFO.
- Uso de la lista tabú: cada vez que generemos una solución con el operador $flip(s, f)$, debemos comprobar si f está en la lista y aceptarla si y sólo si pasa el criterio de aspiración.

2.5. Algoritmo de comparación

Para la comparación de los algoritmos implementados consideraremos el algoritmo voraz *Sequential forward selection*, cuyo pseudocódigo es el siguiente:

La idea es sencilla: en cada iteración escogemos la característica, de entre las aún no seleccionadas, que mejor valor de la función objetivo produce, si y sólo si este valor es mejor que el actual.

Pseudocódigo 6 Algoritmo de comparación

```
1: function sequentialForwardSelection(train, target)
2:     s ← genZeroSolution()
3:     bestScore ← 0
4:     while there was improvement with some feature do
5:         for every feature f in not selected features do
6:             s ← addFeature(s,f)
7:             currentScore ← score(s, train, target)
8:             if currentScore > bestScore then
9:                 bestScore ← currentScore
10:                bestFeature ← f
11:            s ← removeFeature(s,f)
12:        if there was a best feature f then
13:            s ← addFeature(s,f)
14:    return s, bestScore
```

3. Desarrollo de la práctica

La práctica se ha desarrollado por completo en Python, definiendo cada algoritmo en una función diferente con cabeceras similares —mismo número y tipo de parámetros— con el fin de poder automatizar el proceso de recogida de datos.

3.1. *Framework* de aprendizaje automático

Se ha usado, además, el módulo *Scikit-learn*, del que se han usado las siguientes funcionalidades:

- Particionamiento de los datos. *Scikit-learn* aporta una función para hacer un particinado aleatorio de los datos en una parte de aprendizaje y otra de test. Esto se ha usado para implementar la téxnica 5×2 *cross-validation*.
- Evaluación de la función objetivo. *Scikit-learn* tiene implementado el algoritmo de los k vecinos más cercanos, además de la técnica del *leave-one-out*, usada en la evaluación de cada una de las soluciones consideradas para cada algoritmo

3.2. Manual de usuario

Para la ejecución de la práctica es necesario tener instalado Python 3 y el módulo *Scikit-learn*.

Todo se encuentra automatizado en el fichero *characteristicSelection.py*, así que sólo es necesario ejecutar la siguiente orden desde el directorio raíz de la práctica *python characteristicSelection.py*

Así se ejecutarán todos los algoritmos con todas las bases de datos usando la ténica del 5×2 *cross-validation*. Las tablas generadas se guardarán en el directorio *results*.

La semilla utilizada se inicializa al principio de la ejecución del programa con la línea *np.random.seed(19921201)*

4. Análisis de resultados

En esta sección vamos a presentar los datos recogidos de la ejecución de todos los algoritmos con las tres bases de datos consideradas: *WDBC*, *Movement Libras* y *Arrhytmia*. Las bases de datos se han considerado completas en todos los casos, tal y como se nos entregaron —arreglando alguna columna defectuosa y homogeneizando el nombre de la columna de clasificación para poder automatizar el proceso—.

Para el análisis de cada algoritmo con cada base de datos se han generado cinco particiones aleatorias de los datos y se ha ejecutado el algoritmo considerando cada partición como datos de entrenamiento y test, con la técnica 5×2 *cross-validation*.

En cada una de estas ejecuciones se han medido los siguientes datos:

- Tasa de clasificación en la partición de entrenamiento —en %—.
- Tasa de clasificación en la partición de test —en %—.
- Tasa de reducción de las características —en %—.
- Tiempo de ejecución —en segundos—.

Veamos ya los datos y analicemos los resultados obtenidos:

4.1. Clasificador k-NN

Cuadro 1: Datos del clasificador k-NN

En la tabla 1 se pueden ver los datos obtenidos del clasificador k-NN. La selección de características en este algoritmo es nula, ya que es la propia función objetivo considerando la totalidad de las características. Aún así, se ha añadido aquí para conocer la tasa de clasificación en los conjuntos de entrenamiento y de test considerando como solución la trivial: esto es, todas las características.

Como vemos, aunque en la primera base de datos las tasas de clasificación son buenas, en las otras dos son muy mejorables, lo que nos da una idea de la necesidad de la reducción de características.

4.2. Algoritmo de comparación

En la tabla 2 vemos los resultados del algoritmo de comparación, el *Sequential forward selection*. Este algoritmo voraz tiene una alta tasa de reducción de características, pero la tasa de clasificación no mejora la del clasificador con la solución trivial.

Particiones	WDBC				Movement Libras				Arrythmia			
	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,8873	9,543,8596	83,3333	0,2376	81,1111	6,777,7778	91,1111	0,9031	72,9167	7,010,3093	98,9209	1,1452
Partición 1-2	97,5439	9,330,9859	80	0,2787	75	6,777,7778	93,3333	0,6818	74,2268	6,302,0833	99,2806	0,8345
Partición 2-1	96,4789	9,508,7719	90	0,1501	81,1111	7,166,6667	88,8889	1,1382	82,8125	7,268,0412	96,7626	3,4327
Partición 2-2	98,2456	9,683,0986	83,3333	0,2358	77,2222	7,277,7778	92,2222	0,7911	79,8969	6,406,25	97,8417	2,2055
Partición 3-1	98,2394	9,508,7719	80	0,2777	74,4444	7,555,5556	90	1,0206	79,6875	7,474,2268	97,1223	3,003
Partición 3-2	96,8421	9,084,507	90	0,1503	78,3333	7,722,2222	84,4444	1,6339	76,2887	6,875	98,5612	1,6115
Partición 4-1	96,831	9,403,5088	83,3333	0,2381	76,1111	7,000	92,2222	0,7981	75,5208	6,752,5773	97,1223	3,0296
Partición 4-2	97,5439	9,577,4648	80	0,279	78,8889	7,500	86,6667	1,3859	84,5361	7,447,9167	96,0432	4,2178
Partición 5-1	97,8873	9,508,7719	80	0,2781	73,3333	6,833,3333	91,1111	0,9061	83,3333	6,855,6701	97,482	2,5733
Partición 5-2	98,5965	9,471,831	83,3333	0,2373	80,5556	7,333,3333	88,8889	1,1402	76,8041	6,822,9167	97,8417	2,1995
Medias	97,6096	9,462,1572	83,3333	0,2363	77,6111	7,194,4444	89,8889	1,0399	78,6023	6,921,4991	97,6978	2,4253

Cuadro 2: Datos del algoritmo *Sequential forward selection*

Esto se debe a que consideramos cada característica de una forma secuencial, y una vez seleccionamos una, es imposible descartarla. Aún así, este algoritmo podría ser interesante si lo que buscamos es una reducción drástica del número de características —como vemos, sobre le 80 %— sin perder mucha información —las tasas de clasificación son más o menos iguales a las del clasificador con la solución trivial—.

4.3. Búsqueda local primero el mejor

Cuadro 3: Datos de la búsqueda primero el mejor

En la tabla 3 vemos los datos de la primera metaheurística real considerada: la búsqueda local primero el mejor.

Esta metaheurística consigue unas tasas de clasificación algo mejores que en los casos anteriores y, sobre todo, es muchísimo más rápida que el algoritmo SFS.

Esto se debe a que es un algoritmo que aglutina la naturaleza casi voraz del SFS pero atendiendo a criterios mucho más sensatos. Vemos así cómo la búsqueda en el entorno de soluciones, generando vecinos y usando algún criterio para seleccionarlos —en este caso, el que sea mejor de entre los vecinos— es una buena estrategia —sobre todo en tiempo— para este problema.

4.4. Enfriamiento simulado

Cuadro 4: Datos del enfriamiento simulado

En la tabla 4 se encuentran los datos referentes a la ejecución del enfriamiento simulado sobre todas las bases de datos.

Vemos cómo conseguimos una tasa de clasificación fuera de la muestra algo mejor que en el algoritmo anterior, aunque los tiempos ahora se disparan.

La tasa de reducción, sin embargo, es también mayor, así que si buscamos una reducción que permita acelerar futuros procesos de aprendizaje —no olvidemos que el objetivo de nuestro problema es facilitar el trabajo ed algoritmos de aprendizaje posteriores— y un aumento en la tasa de clasificación, aunque pequeño, es altamente valorado, este algoritmo es el mejor de los que hemos visto hasta ahora.

Sin embargo, si el tiempo es una restricción muy grande, la búsqueda local es una solución mucho mejor

4.5. Búsqueda tabú básica

Cuadro 5: Datos de la búsqueda tabú básica

En la tabla 5 vemos los datos de la última metaheurística considerada: la búsqueda tabú básica.

Lo primero que llama la atención es el tiempo usado en la ejecución. A este algoritmo no se le han añadido más condiciones de parada que llegar al número máximo de evaluaciones, así que se tienen que recorrer 15000 soluciones, además de mantener la lista tabú y hacer todas las comprobaciones necesarias. Es un algoritmo computacionalmente costoso.

Los resultados, además, no son mucho mejores a los anteriores. Si consideramos, por ejemplo, la base de datos *WDBC* vemos que el coste de pasar de algo más de un minuto a más de una hora proporciona una tasa de clasificación sólo 0,17 puntos mejor.

La tasa de reducción sí mejora algo más en este caso, así que si esta reducción va a tener un impacto muy grande en el algoritmo de aprendizaje posterior —probablemente incluso más costoso que este—, esta metaheurística puede ser considerada.

4.6. Datos generales

Cuadro 6: Datos generales

En la tabla 6 vemos un resumen de todos los datos obtenidos tras las ejecuciones de las metaheurísticas con las bases de datos.

Vemos ahora más claro el coste computacional de la búsqueda tabú, varias veces más grande que cualquier de los otros algoritmos. La reducción de características en el SFS es otro dato que llama la atención: no debe sorprendernos,

sin mebargo, ya que al ir escogiendo las características secuencialmente, es difícil que alguna no añada algo de mejora —salvo al final, cuando ya se han descartado las características malas o posiblemente ruidosas—.

Un último aspecto a edstacar es la poca diferencia en la tasa de clasificación fuera de la muestra, que es la que realmente nos interesa. Es normal, sin embargo, ya que el espacio de búsqueda es extremeadamente grande y, aunque la búsqueda sea mucho más exhaustiva, nada nos garantiza conseguir soluciones mucho mejores.

Sin embargo, hay que tener siempre en cuenta que este es un paso previo para algoritmos de aprendizaje, así que cualquier mejora, por pequeña que sea, puede derivar en una gran reducción del tiempo en y aumento del éxito en los algoritmos posteriores.